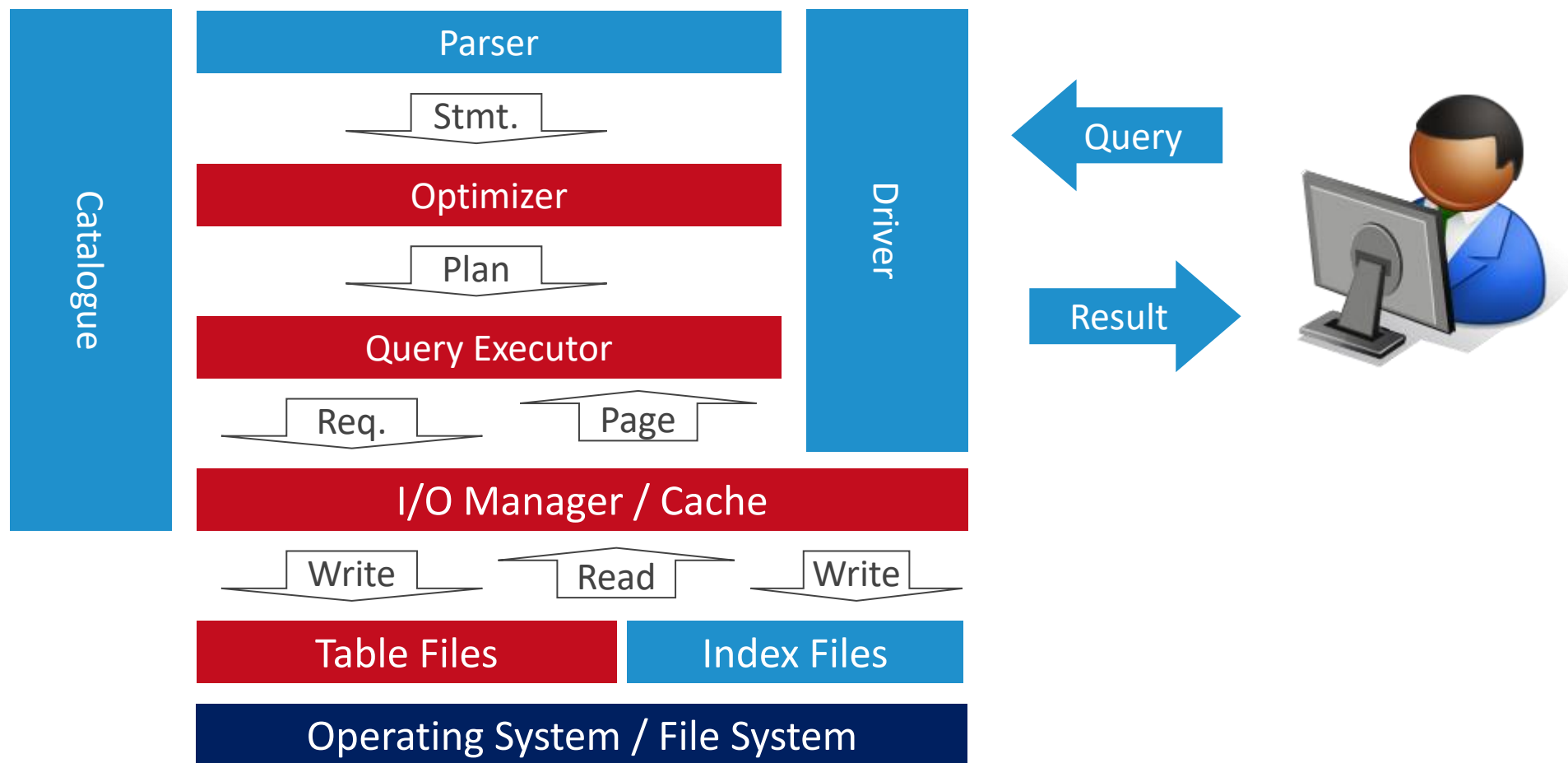


DBTLAB – Exercise 5: Query Execution – Operators II

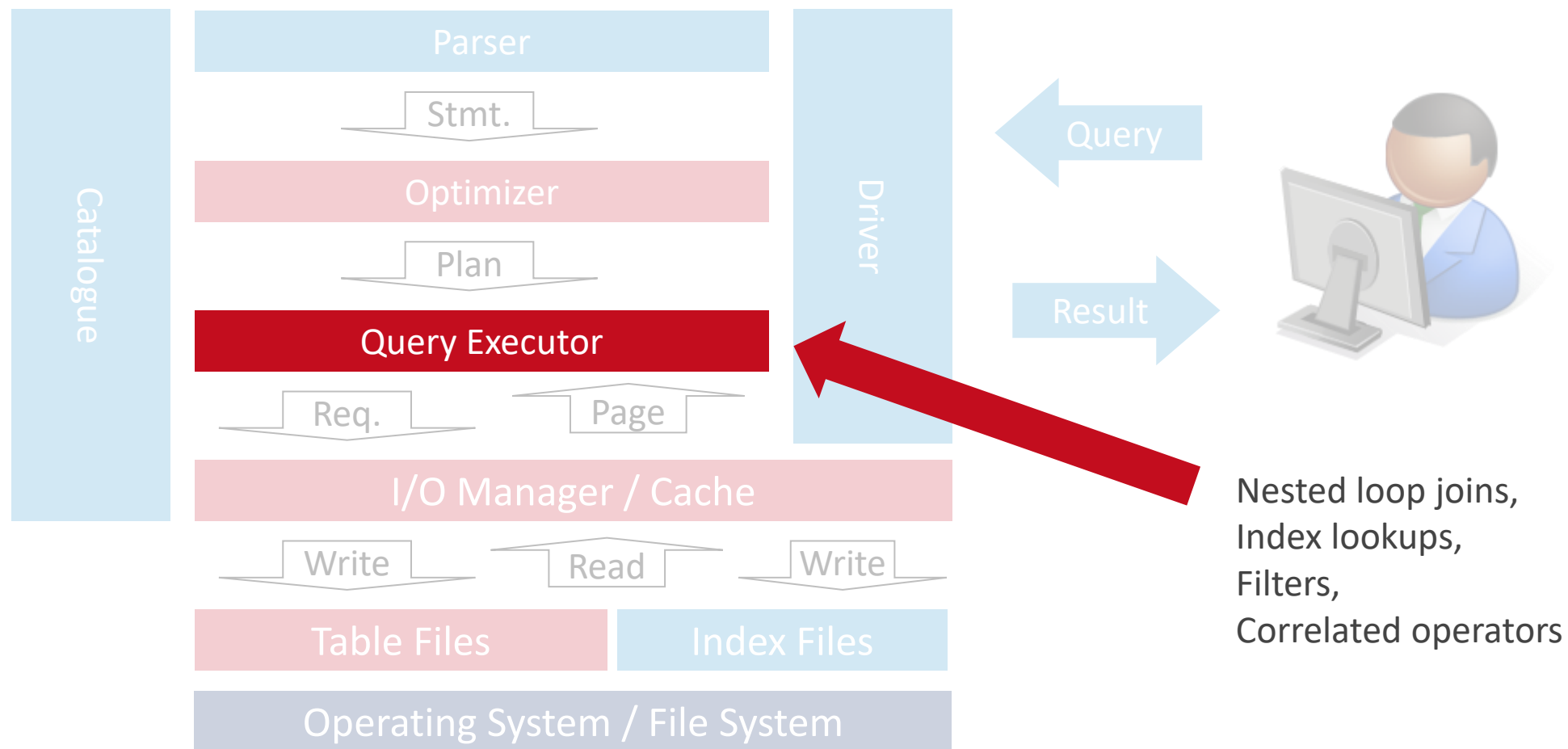
Rudi Poepel Lemaitre

based on material from Volker Markl, Viktor Rosenfeld, Jonas Traub, Martin Kiefer

Basic system architecture



Where are we today?



Last lecture

- Query execution models
 - **Iterator model**
 - Bulk processing
 - Query compilation
- Iterator model operator API
 - open
 - next
 - close
- Table scan and index scan operators

This exercise

- Operators to implement nested loop joins and additional predicates in the WHERE clause.
- NestedLoopJoinOperator
 - Nested loop join
 - Indexed nested loop join
- IndexLookupOperator
 - Retrieve the RID of a matching tuple based on a predicate.
- IndexCorrelatedLookupOperator
 - Retrieve the RID of a matching tuple based on a correlated tuple.
- FetchOperator
 - Fetch the columns of a tuple retrieved through an index.
- FilterOperator
 - Filter tuples based on a predicate.
- FilterCorrelatedOperator
 - Filter tuples based on a correlated tuple.

Background: Why have different join implementations?

- The simplest join is the nested loop join.
 - Complexity: $O(n*m)$
- Use index to lookup inner tuple.
 - The index will only give us a matching RID.
 - We still have to fetch the complete tuple to produce the join result.
 - What if the join condition matches a lot of tuples?
- Use a sort merge join.
 - Sorting: $O(n * \log n) + O(m * \log m)$
 - Merging: $O(n + m)$
 - What if the data is already sorted?
 - What if we have to sort to disk?
- Use a hash join:
 - Expected complexity: $O(n + m)$
 - What about inequality joins?
- The fastest join algorithm depends on the data and other operators in the query plan.

```
for outerTuple in outerTable:
    for innerTuple in innerTable:
        if outerTuple and innerTuple match join condition:
            output pair of outerTuple and innerTuple
```

Join operators in mini-dbs

- Simple nested loop join
- Index nested loop join
- Sort merge join
 - Next exercise

Correlated subplans

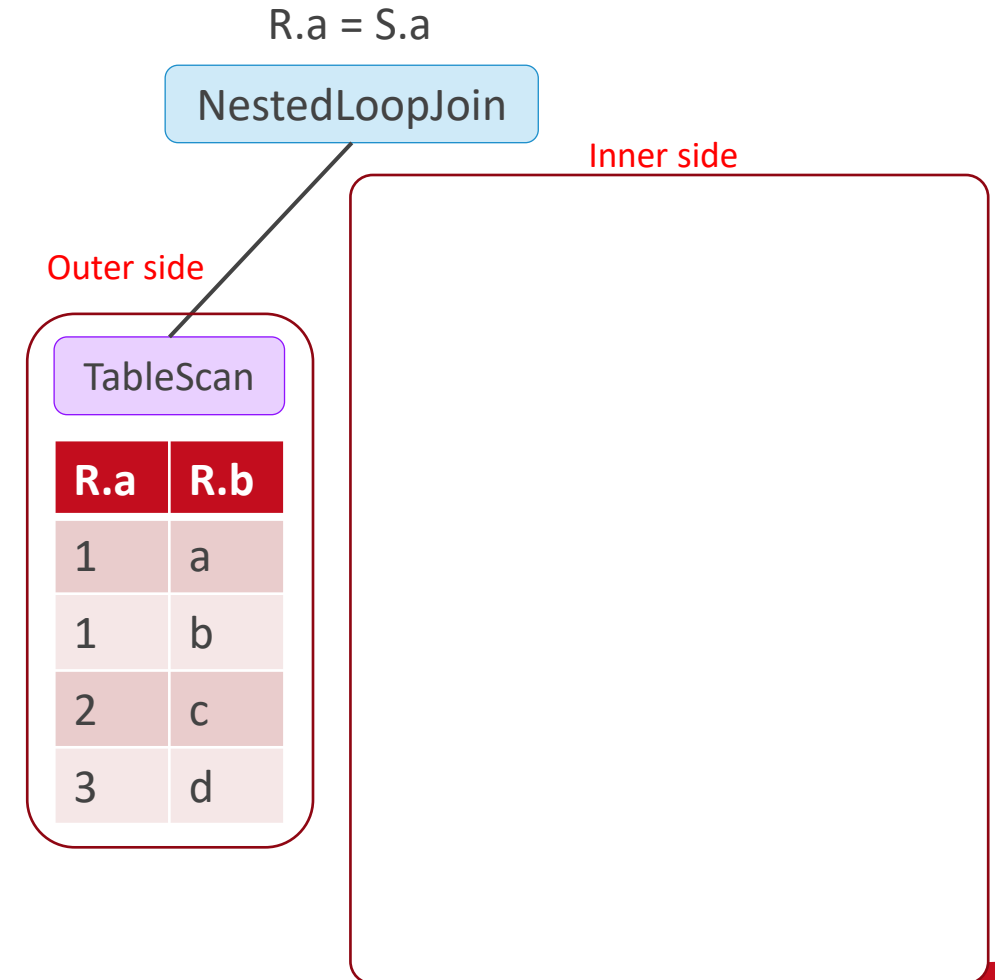
- The method `open()` takes a *correlated tuple* as a parameter.
- General case: An operator (and therefore the subplan of this operator) is opened once, then the `next()` method is repeatedly called, then the operators is closed.
 - Especially true for the root operator of the query plan.
- Some operators are opened and closed several times during the execution of a query.
 - The tuples produced by `next()` depend on the *correlated tuple* specified in `open()`.
- Our system: The operator on the inner side of a nested loop join.
 - Opened for each tuple from the outer table to find matching tuples in the inner table.
 - The tuple from the outer table is the correlated tuple.
 - That means all operators in the subplan of the inner child are also opened/closed several times.
 - The entire subplan is correlated.

Correlated tuples

- Act as a filter to the tuples returned by the correlated subplan.
- Evaluated directly by the operator.
 - IndexCorrelatedLookupOperator: Use the tuple as lookup key.
 - FilterCorrelatedOperator: Use the tuple field as a filter predicate.
- Passed to child operators.
 - FetchOperator: Pass information to a IndexCorrelatedLookupOperator child.
 - Children of the inner side of nested loop joins.

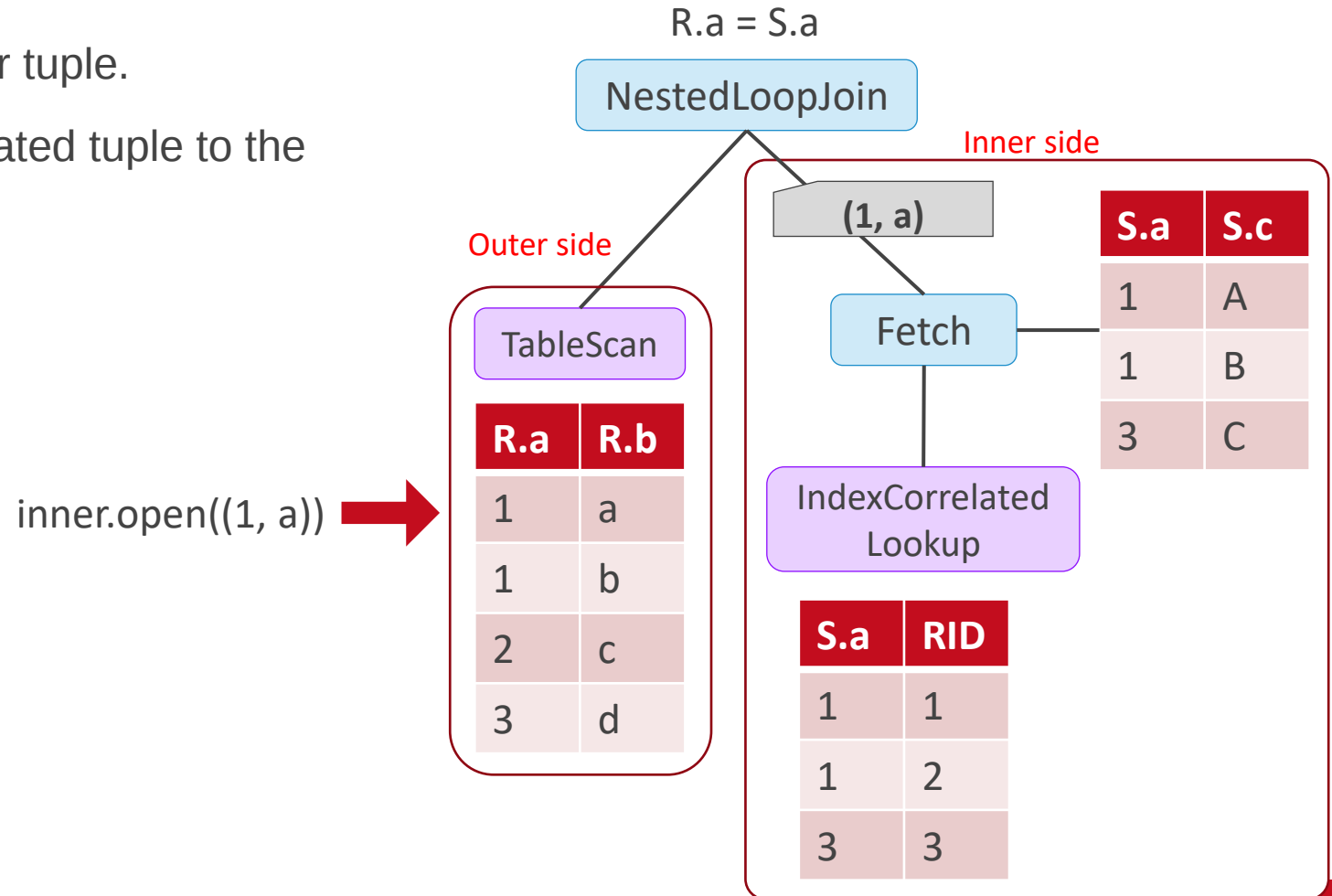
Correlated plan example

- The inner plan is opened for each outer tuple.
- The outer tuple is passed as the correlated tuple to the inner plan.



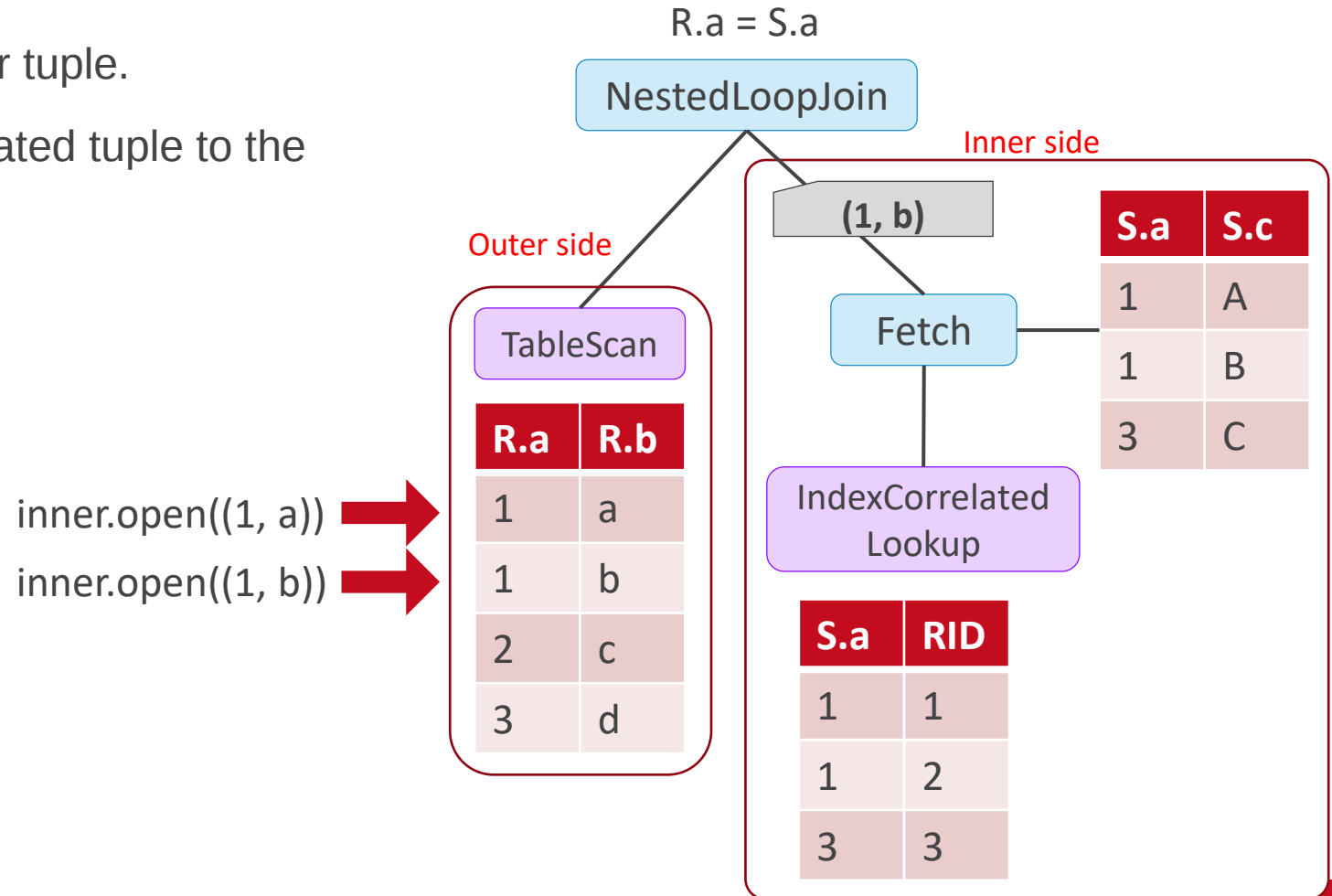
Correlated plan example

- The inner plan is opened for each outer tuple.
- The outer tuple is passed as the correlated tuple to the inner plan.



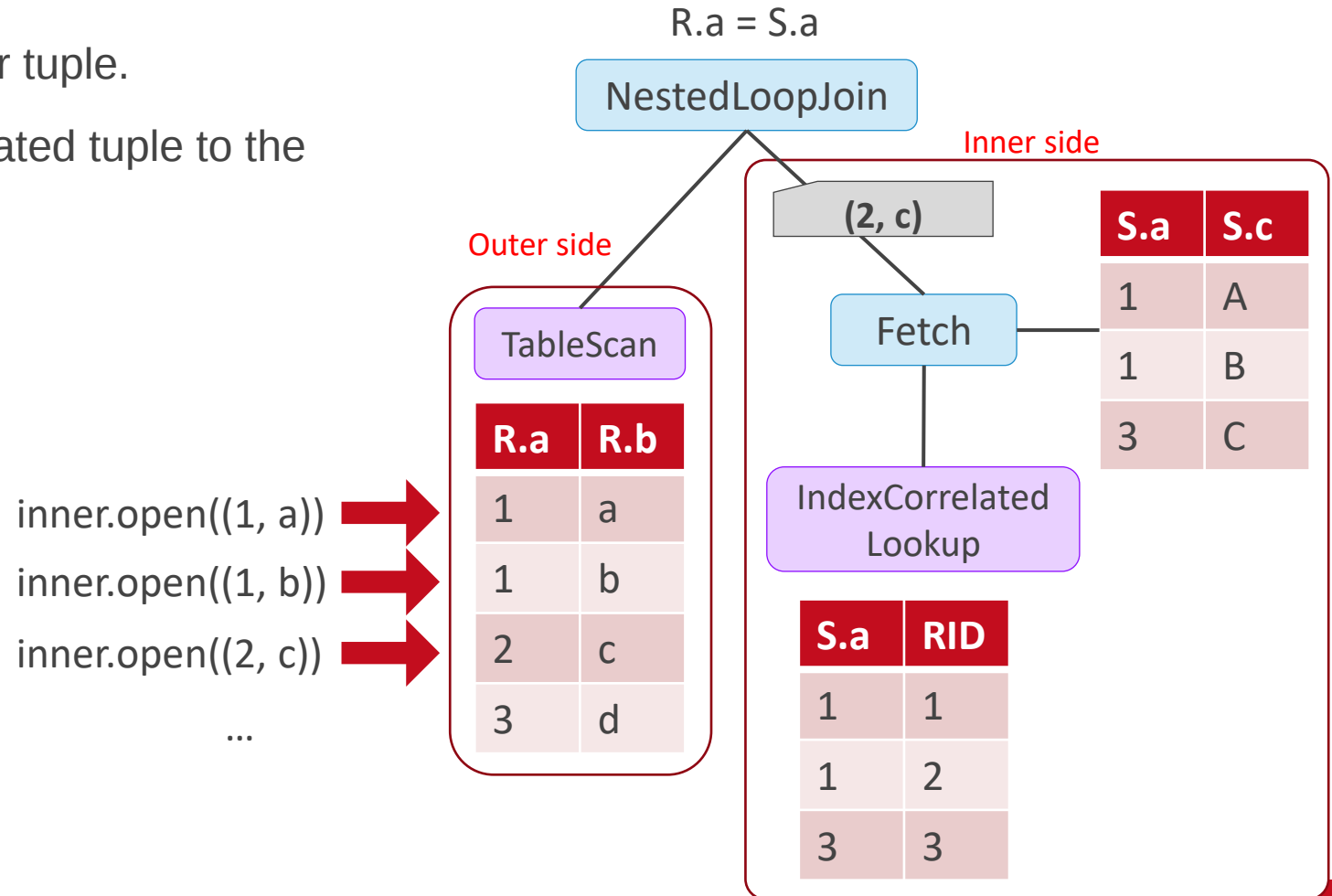
Correlated plan example

- The inner plan is opened for each outer tuple.
- The outer tuple is passed as the correlated tuple to the inner plan.



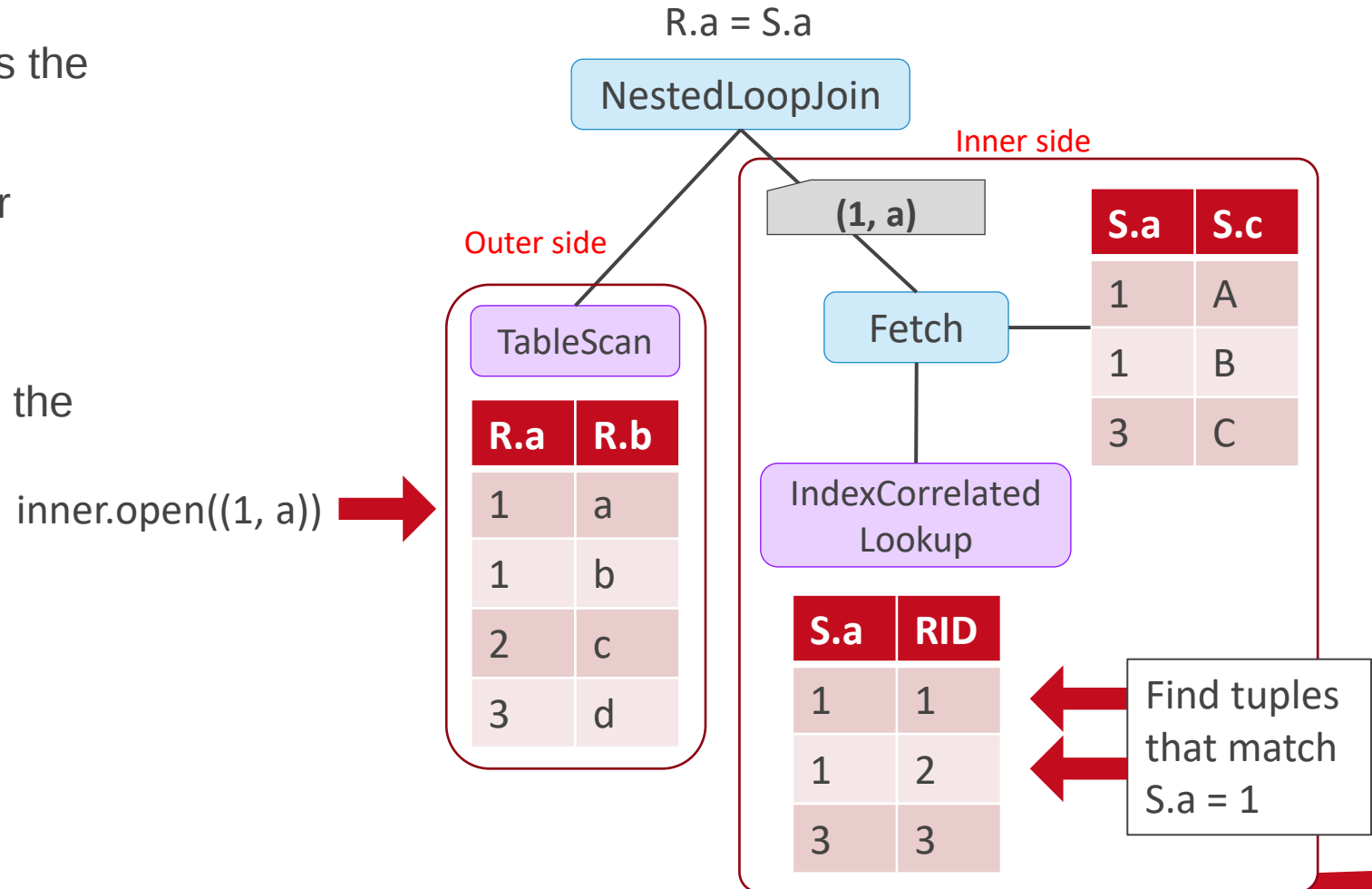
Correlated plan example

- The inner plan is opened for each outer tuple.
- The outer tuple is passed as the correlated tuple to the inner plan.



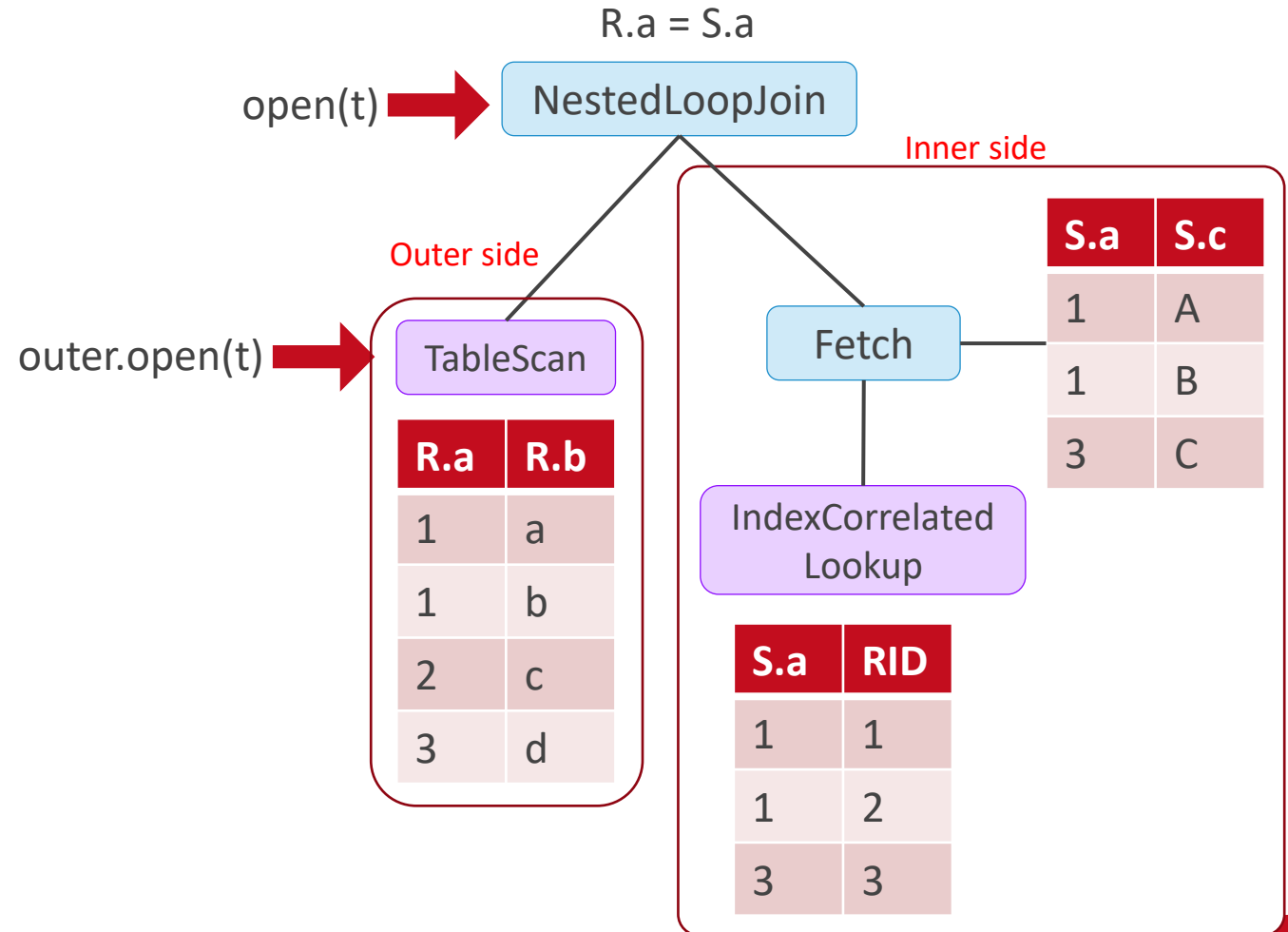
Correlated plan example

- The IndexCorrelatedLookupOperator uses the correlated tuple to find matching RIDs.
- The FetchOperator has to pass the inner correlated tuple to the IndexCorrelatedLookupOperator.
- The join predicate is evaluated through the correlated tuple.



Correlated plan example

- The NestedLoopJoin could be the inner child of another join.
- It is opened with its own correlated tuple.
- This is passed to the outer child.
- Implicitly filters the looked up tuples in the inner child.



Operators

NestedLoopJoinOperator

- Used to implement both types of nested loop joins.
 - Simple nested loop join: Children are accessed through TableScanOperator.
 - Indexed nested loop join: One child is accessed through an IndexCorrelatedLookupOperator.
- next() always returns a single joined tuple (or null if finished).
 - Might have to close the inner table, advance to the next outer tuple, and open the inner table again.
 - Might have to advance multiple child tuples until the join predicate is true.
- Factory method: createNestedLoopJoinOperator
 - PhysicalPlanOperator outerChild, PhysicalPlanOperator innerChild
 - Inner and outer child subplans
 - JoinPredicate joinPredicate
 - Optional join predicate
 - **int[]** columnMapOuterTuple, **int[]** columnMapInnerTuple
 - Projected columns from the child tables.

Join predicates

- Case 1: Join predicate is already evaluated through the correlated tuple.
 - Parameter joinPredicate is null.
- Case 2: Inner child does not evaluate the correlated tuple, e.g., a TableScanOperator.
 - Use joinPredicate.evaluate() on the current outer and inner tuples.

Output projection

- The output tuple of a nested loop join is a projection of some attributes from the outer and some attributes from the inner tuple.
- These are specified in the `columnMapOuterTuple` and `columnMapInnerTuple` arrays.
- Same length and indicate the length of the output tuple.
- Array positions encode the attribute positions in the output tuple.
 - A value -1 indicates that the other input tuple is used to fill the attribute of the output tuple.
 - A value other than -1 indicates that the corresponding field of the input tuple should be used as the field in the output tuple that corresponds to the array position.

Output projection example

- Base tables
 - CUSTOMER (c_custkey, c_name, c_address, c_nationkey, c_phone, c_acctbal, c_mktsegment, c_comment)
 - NATION (n_nationkey, n_name, n_regionkey, n_comment)

– Query `SELECT c_name, c_address, n_name, c_phone FROM customer JOIN nation ON c_nationkey = n_nationkey`

- columnMapOuterTuple for table NATION
- columnMapInnerTuple for table CUSTOMER

Index	0	1	2	3
	-1	-1	1	-1
	1	2	-1	4

- Output tuple structure

Output projection example

– Base tables

- **CUSTOMER** (c_custkey, c_name, c_address, c_nationkey, c_phone, c_acctbal, c_mktsegment, c_comment)
- **NATION** (n_nationkey, n_name, n_regionkey, n_comment)

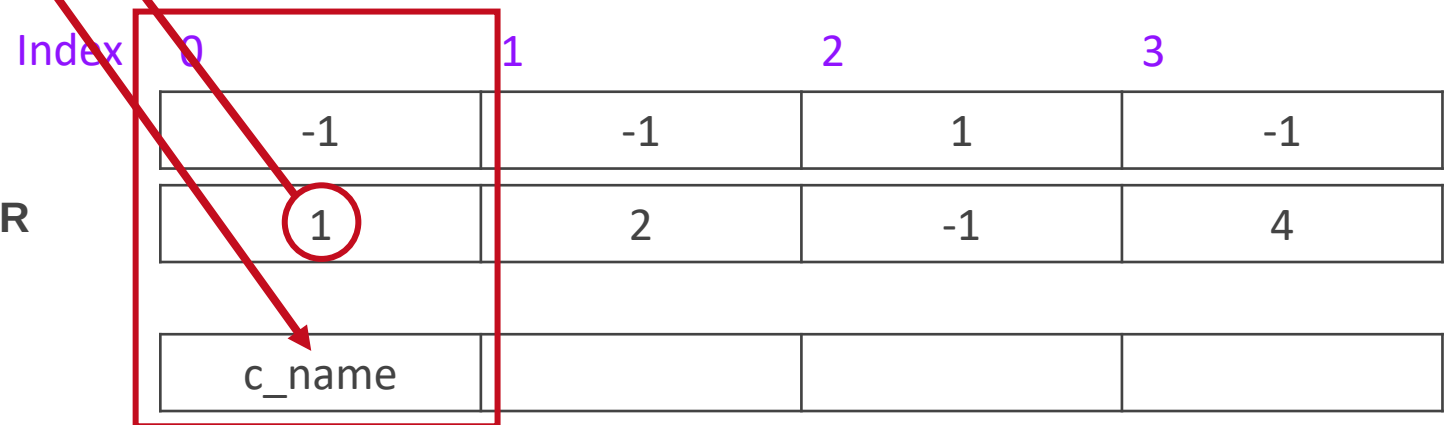
– Query

```
SELECT c_name, c_address, n_name, c_phone FROM customer JOIN nation
ON c_nationkey = n_nationkey
```

- columnMapOuterTuple for table **NATION**
- columnMapInnerTuple for table **CUSTOMER**
- Output tuple structure

Index

0	1	2	3
-1	-1	1	-1
1	2	-1	4
c_name			



Output projection example

– Base tables

- **CUSTOMER** (c_custkey, c_name, c_address, c_nationkey, c_phone, c_acctbal, c_mktsegment, c_comment)
- **NATION** (n_nationkey, n_name, n_regionkey, n_comment)

– Query

```
SELECT c_name, c_address, n_name, c_phone FROM customer JOIN nation
ON c_nationkey = n_nationkey
```

– columnMapOuterTuple for table **NATION**

– columnMapInnerTuple for table **CUSTOMER**

– Output tuple structure

Index 0

Index 1			
2	3		
-1	-1	1	-1
1	2	-1	4
c_name	c_address		

Output projection example

– Base tables

- **CUSTOMER** (c_custkey, c_name, c_address, c_nationkey, c_phone, c_acctbal, c_mktsegment, c_comment)
- **NATION** (n_nationkey, n_name, n_regionkey, n_comment)

– Query

```
SELECT c_name, c_address, n_name, c_phone FROM customer JOIN nation
ON c_nationkey = n_nationkey
```

– columnMapOuterTuple for table **NATION**

– columnMapInnerTuple for table **CUSTOMER**

– Output tuple structure

Index 0

1

2

3

-1	-1	1	-1
1	2	-1	4
c_name	c_address	n_name	

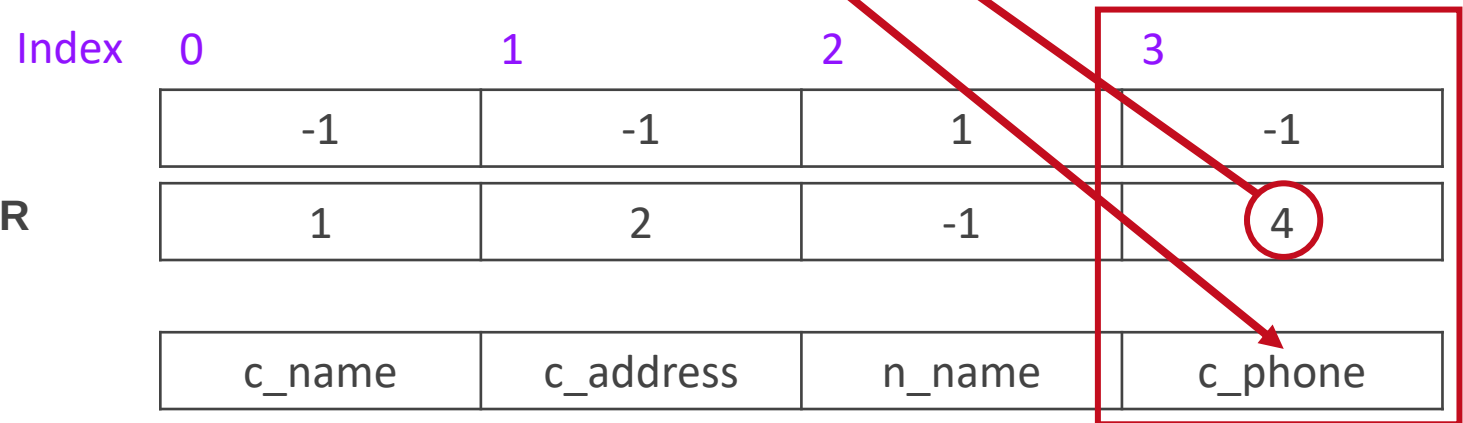
Output projection example

- Base tables
 - **CUSTOMER** (c_custkey, c_name, c_address, c_nationkey, c_phone, c_acctbal, c_mktsegment, c_comment)
 - **NATION** (n_nationkey, n_name, n_regionkey, n_comment)

– Query `SELECT c_name, c_address, n_name, c_phone FROM customer JOIN nation ON c_nationkey = n_nationkey`

- columnMapOuterTuple for table **NATION**
- columnMapInnerTuple for table **CUSTOMER**
- Output tuple structure

Index	0	1	2	3
	-1	-1	1	-1
	1	2	-1	4
	c_name	c_address	n_name	c_phone



IndexLookupOperator

- Look up a key or a key range in an index.
 - Equality predicate: WHERE city = 'Berlin'
 - Between predicate: WHERE age >= 18 AND age <= 90
- Returns tuples with exactly one column, the RID of matching tuples.
 - IndexScanOperator from last lecture returns the keys and not the RIDs.
- Occurs together with FetchOperator as an alternative to a TableScanOperator.
- Factory method: getIndexLookupOperator
 - BTreeIndex index
 - The B+ tree index
 - DataField equalityLiteral
 - Value for an equality comparison
- Factory method: getIndexScanOperatorForBetweenPredicate
 - BTreeIndex index
 - DataField lowerBound, **boolean** lowerIncluded, DataField upperBound, **boolean** upperIncluded
 - Lower and upper boundaries of the between predicate

IndexCorrelatedLookupOperator

- Look up keys in an index based on the correlated tuple passed in `open()`.
- Returns tuples with exactly one column, the RID of matching tuples.
- Used as inner table in an index nested loop join.
- Only used for equality joins.
- Factory method: `getIndexCorrelatedScanOperator`
 - BTreeIndex index
 - **int** `correlatedColumnIndex`
 - Column in the correlated tuple which contains the key that should be looked up.

FetchOperator

- Takes an RID and fetches the referenced tuple from a table.
 - RIDs are received from the child operator, i.e., IndexLookupOperator or IndexCorrelatedLookupOperator.
- Factory method: createFetchOperator
 - PhysicalPlanOperator child
 - BufferPoolManager bufferPool
 - Buffer pool to obtain pages from
 - **int** tableResourceId
 - The resource ID of the table that is used to retrieve pages from the buffer pool.
 - **int[]** outputColumnMap
 - The columns that should be fetched.
 - See TableScanOperator from last lecture.

FilterOperator

- Filter tuples from the child operator according to a predicate.
- Passes the correlated tuple to the child operator.
- Factory method: createFilterOperator
 - PhysicalPlanOperator child
 - LocalPredicate predicate
 - The predicate which evaluates the filter.

FilterCorrelatedOperator

- Filter tuples from the child operator according to a predicate and the correlated tuple.
- Does not pass the correlated tuple to the child operator.
- Factory method: createCorrelatedFilterOperator
 - PhysicalPlanOperator child
 - JoinPredicate correlatedPredicate
 - A predicate which compares the current tuple of the child against the correlated tuple passed in open().

Tests and points

- NestedLoopJoinOperator (NLJ)
- IndexLookupOperator (IL)
- FetchOperator (Fetch)
- IndexCorrelatedLookupOperator (ICL)
- FilterOperator (Filter)
- FilterCorrelatedOperator (FC)

Test name	Points	Operators
testNestedLoopJoinScanOnly	1.5	NLJ
testIndexNestedLoopJoin	1.5	NLJ, ICL, Fetch
testIndexNestedLoopJoinWithFilter	1.5	NLJ, ICL, Fetch, Filter
testMultiwayJoin	1.5	NLJ, ICL, Fetch, Filter, FC
testUncorrelatedIndexEqualityPredicate	1.5	IL
testUncorrelatedIndexRangeFetch	1.5	IL, Fetch

Multiway join

- The multi way join creates the following query plan
- NestedLoopJoinOperator testTopNLJNOperator
 - NestedLoopJoinOperator testTopNLJNOperator
 - NestedLoopJoinOperator testSecondJoinOp
 - TableScanOperator testFirstTableScan
 - TableScanOperator testSecondTableScan
 - TableScanOperator testThirdTableScan
 - NestedLoopJoinOperator testCorrelatedNLJNOperator
 - FilterCorrelatedOperator testFourthFilter
 - TableScanOperator testFourthScan
 - FilterOperator testFifthFilter
 - FetchOperator testFifthFetch
 - IndexCorrelatedLookupOperator testIndexScan
 - BTreeIndex testFifthIndex